ESD-TR-71-342 AD 393376

A GENERALIZED COMPACTIFYING GARBAGE COLLECTOR (A COMPUTER STORAGE MANAGEMENT ALGORITHM)

Ben Wegbreit

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

(Prepared under Contract No. F19628-68-C-0379 by Harvard University, Cambridge, Massachusetts 02138.)

[illegible]

A GENERAL MODEL COMPACTIFYING [illegible] IN COMPUTER STORAGE MANAGEMENT ALGORITHMS

Ben Wegbreit

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
(ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

(Prepared under Contract No. F19628-[illegible]-C-0379 by Harvard University,
Cambridge, Massachusetts 02138.)

NOTICE

[illegible]

[illegible]

ESD-TR-71-342

A GENERALIZED COMPACTIFYING GARBAGE COLLECTOR (A COMPUTER STORAGE MANAGEMENT ALGORITHM)

Ben Wegbreit

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

(Prepared under Contract No. F19628-68-C-0379 by Harvard University,
Cambridge, Massachusetts 02138.)

FOREWORD

This report presents the results of research conducted by Harvard University, Cambridge, Massachusetts in support of ARPA Order 952 under contract F19628-68-C-0379. Dr. John B. Goodenough (ESD/MCDT-1) was the ESD Project Monitor.

This technical report has been reviewed and is approved.

[illegible]

EDMUND P. GAINES, JR., Colonel, USAF
Director, Systems Design & Development
Deputy for Command & Management Systems

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Harvard University Center for Research in Computing Technology Cambridge, Mass. 02138	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP N/A

3. REPORT TITLE
A GENERALIZED COMPACTIFYING GARBAGE COLLECTOR
(A COMPUTER STORAGE MANAGEMENT ALGORITHM)

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
None

5. AUTHOR(S) *(First name, middle initial, last name)*
Ben Wegbreit

6. REPORT DATE August 1971	7a. TOTAL NO. OF PAGES 22	7b. NO. OF REFS 8

8a. CONTRACT OR GRANT NO. F19628-68-C-0379	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-71-342
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

A technique for compactifying garbage collection is presented. The method is applicable to very general classes of nodes, works even when pointers point into the middle of nodes, and preserves arbitrarily complex re-entrant pointer structures.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	W
Storage compactification						
Garbage collection						
Compactifying garbage collection						
Storage allocation						
List processing						

UNCLASSIFIED
Security Classification

ABSTRACT

A technique for compactifying garbage collection is presented. The method is applicable to very general classes of nodes, works even when pointers point into the middle of nodes, and preserves arbitrarily complex re-entrant pointer structures.

A GENERALIZED COMPACTIFYING GARBAGE COLLECTOR *

Introduction

Languages providing dynamic storage allocation usually provide for the automatic reclamation of unused storage by garbage collection (e.g., see McCarthy, 1962).

In its simplest form, a garbage collector reclaims unused storage while leaving objects in use _in situ_, i.e., in the same memory locations. Hence, the storage in use eventually becomes scattered throughout memory. Under common conditions, this has deleterious consequences. If blocks of storage may be required in various sizes, the fragmentation of storage may produce a situation where there is no single free block large enough to satisfy some request although the total amount of free storage is great enough. A solution to this problem is _compactifying_ garbage collection. That is, garbage collection in which the storage in use is moved to a contiguous region of memory and all pointers are adjusted to reflect this movement. The resulting list structure has the same topology as the old, so that re-entrancy and sharing of common substructure are preserved.

There are several algorithms for doing this for _particular_ list structures. Algorithms by Minsky (1963) and Fenichel and Yochelson (1969) work for LISP, i.e., list structure represented by linked blocks of two pointers each. Algorithms by Hansen (1969) and Cheney (1970) handle one-way list structure represented as a mixture of linked two-pointer blocks and sequential blocks of list elements. Each of these algorithms is tailored to a particular class of list structure and does _not_ generalize to the case of arbitrary nodes. This paper presents a compactifying garbage collection technique that works on arbitrary

* Also included as number 5-71 in the publication series of the Harvard Center for Research in Computing Technology.

nodes, even when pointers point into the middle of nodes. The next section outlines the need for such a technique.

Motivation

Typically, a list-processing system offers one data representation for all structures. For many purposes, this is quite satisfactory. For others, this is awkward and inefficient. To overcome this, a few languages – e.g., Algol 68 (Wijngaarden, 1969) and EL1 (Wegbreit, 1971) – have been developed which allow the programmer to create his own representations. A programmer might choose to use rings, two-way lists with back as well as front pointers, threaded lists, or a mixture of these and conventional two-pointer nodes. Further, he might choose to have the nodes contain one or more fields containing non-pointer information: integer counts, complex numbers, character strings, bit vectors, etc. In general, the programmer is free to define the sorts of objects he needs, allocate such objects, and link these up as desired. It is useful to ascribe a data type or mode to each sort of object.

The set of modes permitted by such languages may be described recursively as including:

1) a fixed set of primitive modes such as integer, real, character, and Boolean. The actual set of primitives is immaterial so long as there is a predicate, SIMPLE, which is true of the primitive modes.
2) pointers to objects whose modes are in this set.
3) structures and arrays whose components have modes belonging to this set.

Examples of these are:

1′) an integer; a character
2′) a pointer to an integer; a pointer to an array of Booleans
3′) a structure consisting of ⟨an integer⟩ and ⟨a pointer to a structure like this one⟩; a structure consisting of ⟨a real⟩ and ⟨a structure

consisting of an integer and an array of characters⟩.

Since we deal with many different sorts of objects, techniques for list structure garbage collection carry over only partially and compactification techniques do not carry over at all. A general purpose garbage collector must be able to

(A) deal with nonuniform sized objects

(B) trace a pointer embedded in a substructure of an object

(C) handle situations in which several pointers separately reference both an object and one or more components.

As an illustration of the last problem, it may be that in the last example above, one pointer references the entire major structure, while another references the real, and another references just the fourth character in the character array.

There are a variety of ways of simplifying the problem, but at unacceptable expense.† One can, for example, handle (A) by prefixing each object which can

†Indeed, one can assume the existence of a very large virtual memory in which compactification is employed principally to increase the percentage of useful information on each page (i.e., to reduce page faults). In such a situation, one can reserve a second region as large as the one to be compacted and perform compactification by copying from one to the other. Hence, compactification is straightforward. This paper is concerned specifically with the case in which the address space is limited and reservation of such a second region is unacceptable.

be pointed to with a type or length code. Similarly, compactification can be made easy if each object is prefixed by a backpointer to a ring of all those pointers which point to the object. Alternatively, one can reserve room for the backpointer but construct the ring only during garbage collection. However, these simplifications require extra garbage collection fields in each object that can be pointed to – a price which is unacceptable in view of (C) above. The technique to be described does not require special garbage collection fields.

The Technique

A non-compactifying garbage collector has two phases: (1) trace through all reachable structures and mark objects in use, and (2) sweep linearly through memory collecting the unmarked words. To compactify, phase 2 is replaced by: (2') plan where each block of storage in use is to be moved to, (3') adjust each pointer in use to point to the planned address of the object it references, and (4') move the contents of each block in use to its planned address.

Since this is more complicated than normal garbage collections, it is useful to mix the two strategies. That is, most garbage collections are non-compactifying; the compactifier is used as an option in garbage collection, to be invoked only when necessary. In view of the relative infrequency of its use, the compactifying garbage collector need not be overly concerned with speed. It must, however, make a minimum of demands on storage. It is these two considerations that shape the algorithm presented in this paper.

At the point garbage collection is initiated, there is some active environment. This typically includes the stack, the machine registers, and certain fixed locations. The environment specifies the set of base points from which tracing begins and hence defines all the structures active at the time garbage collection is initiated. The outermost loop of the garbage collection trace phase is to consider, in turn, each of the base points and initiate a trace from that point. Since the environment depends on the language and the implementation, we neglect this outer loop and discuss the action for a single base point P. We assume that for any base point, one can determine the sort of object it points to. (Typically, this is implemented by storing with each base point both an address and a type code for the object pointed to.)

The function of the trace phase is to mark all objects in use. This becomes complicated where only part of an object is referenced. Given a pointer to an object, there is no way to know whether the object is part of a larger object and what that larger object is. Further, since objects as small as a bit can be pointed to, there is a problem of how to do the marking. The method used here is to fix on some minimum block size, a word in most machines, and mark to the level of refinement but no finer. Hence, if any bits of a word are in use, the word will be regarded as in use. The choice of quantum unit for marking cannot, however, be altogether arbitrary; there is one requirement which must be met. We discuss this after presenting the trace algorithm.

Unlike list-structure systems, there is no way here to do the marking in the machine word itself. A word may contain a bit vector, an integer, or a real number, leaving no room for marking bits. Hence, we use a bit map — a vector of bits, one per word. Address arithmetic maps the i^{th} word of the free storage region onto the i^{th} element of the bit map. It will prove convenient to use the Algol 68 term _heap_ as an abbreviation for "free storage region". It will also be convenient to suppress the existence of the bit map and speak of marking an object, meaning setting bits corresponding to the address(es) of the object.

To express the trace algorithm, we require some conventions for dealing with objects of arbitrary data types. The following is a reasonable set, implementable in almost all systems. An object can be referenced by means of a pair ⟨A, M⟩ where A is an address and M is the mode (i.e., data type) of the object at that address. MARKED(A, M) marks all words of the object referenced by the pair ⟨A, M⟩. It returns _true_ if all words are _already_ marked; otherwise, it returns _false_. OBJ (A, M) is the object at A of mode M. Two generalized functions are applied to arbitrary objects: ADR (obj) is the address of the object, MD (obj) is the mode of the object.† Three other functions are applicable to various classes of objects. For any compound object X, LENGTH(X) is the number of components, and COMPONENT(X,I) is the I^{th} component. For any pointer object X, VAL(X) is the object pointed to. Two functions test for classes of objects.

†Hence, ADR (OBJ (A, M)) = A and MD (OBJ (A, M)) = M for all A and M.

If M is a data type, SIMPLE(M) is true only if that type is primitive, i.e., neither a structure, nor an array, nor a pointer. POINTER(M) is true only of pointers.

While tracing one pointer from a compound object, it is necessary to keep a handle on the object so as to be able to return to it. For this, we use a stack.[†]

[†]Digressionally, it may be noted that the use of a stack (or equivalently, recursion) is mandatory. This is not the case in LISP where it is possible to store all return points in the structure itself by reversing pointers (Schorr and Waite, 1967). This dispenses with the stack but requires one extra garbage collection bit per word used to interpret the trace state on return to an object after having traced either of its pointers. An attempt to apply this technique to generalized garbage collection fares poorly. Instead of a single bit to determine the state of tracing within an object, one needs an integer index (see below). Further, one needs a type code to determine how to interpret an object. There is no room to store this information in the structure being traced; storing this in a map is obviously impractical. Hence, a stack of some sort is required.

PUSH($e_1 \ldots e_n$) pushes the values of $e_1 \ldots e_n$ onto the stack. EMPTY_STACK( ) is true if the stack is empty; otherwise false. POP($N_1 \ldots N_n$) pops the top n items from the stack and assigns these to the variables $N_n \ldots N_1$.

To trace from a base point P, the trace routine initializes A to the address of and M to the mode of the object pointed to by P and executes the following code:

```
LOOP1:  if MARKED(A,M) then goto UP;
LOOP2:  if SIMPLE(M) then goto UP;
        if POINTER(M) then begin A2 ← ADR(VAL(OBJ(A,M)));
                                 M2 ← MD(VAL(OBJ(A,M)));
                                 A ← A2;
                                 M ← M2;
                                 goto LOOP1
                           end;
        I ← LENGTH(OBJ(A,M));
        if I=0 then goto UP;
LOOP3:  if I>1 then PUSH(A,M,I-1);
        A2 ← ADR(COMPONENT(OBJ(A,M),I));
        M2 ← MD(COMPONENT(OBJ(A,M),I));
        A ← A2;
        M ← M2;
        goto LOOP2;
UP:     if EMPTY_STACK() then goto TRACE_DONE;
        POP(A,M,I);
        goto LOOP3
```

The algorithm begins by marking the referenced object. If the object was previously marked, no further action need be taken for it. If the object is SIMPLE (i.e., primitive), no further action need be taken. If the object is a pointer, it is traced. Otherwise, the object must be a compound object — either an array or a structure. A compound object is traced by considering its components in turn: SIMPLE components are ignored, pointer components are traced, and components which themselves are compound objects are handled by recursive decomposition.

Notice what occurs when two pointers, P1 and P2, respectively reference an object and one of its components. If P1 is traced first, then the entire object is marked. Hence, when P2 is traced, the component is found to be marked and tracing of P2 stops. This occurs <u>even if</u> components of the object remain to be considered: a triple for the remaining components has been previously stacked and will be popped in due time.

If, on the other hand, P2 is traced first, then just the words containing the component are marked. When P1 is subsequently traced, the words containing the other components are found (see below) to be unmarked. Hence, the entire object is processed, each component in turn. The component pointed to by P2 will be considered again; however, should this contain a pointer P3, the object pointed to by P3 will be found to be entirely marked, so that tracing from P3 will not be repeated.

One difficulty in the last case may arise if the component referenced by P2 is part of a <u>one-word</u> object which contains a pointer not part of this component. For example, a structure may consist of a character and a pointer; the pointer P2 may reference the first component — the character — only. When P1 is traced, the entire object will be found to be marked so that the object will not be traced. Hence, the pointer (the second component) will never be traced, causing an error. That is, marking the word containing a component masks off any other components in an object; if the other components are pointers, this causes an error.

There are two possible solutions. One is to arrange the layout of compound objects such that if an object (or distinct sub-object) takes one word and contains a pointer, then it contains nothing else. Another is to observe that the choice of the word as the unit of marking is too coarse. To avoid the problem, it is necessary to choose the unit of marking such that a pointer completely fills at least one marking unit. On most machines, a half-word insures this, if pointers are justified. If a pointer crosses marking unit boundaries (e.g., lies in two half-words), both units are marked in tracing. Since it is impossible for a second pointer to fit into the remainder of the second marking unit, tracing proceeds safely. In choosing between the two solutions, there is a trade-off to be made between the size of marking unit and the packing density of compound objects. The choice is ruled by implementation considerations.

One additional point bears attention. The only reason for considering the components of a compound object is to discover any pointer fields it may contain. Hence, the algorithm would still work, and work much faster, if SIMPLE was interpreted to mean "contains no pointers." With this modification, the algorithm marks but otherwise ignores all objects but those containing pointers. Where compound objects containing no pointers are common, this avoids considerable needless computation and is therefore a very significant improvement.

When the trace phase is completed, all words of the heap (i.e., free storage region) that are in use have been marked. A contiguous set of marked words contains at least one and perhaps many distinct objects which must be preserved. This must be carried out for both compactifying and non-compactifying garbage collection. At this point, the two collection algorithms diverge. The non-compactifying garbage collector forms a free-list[†] consisting of linked

†While it is possible to use a simple free-list, it is far better to modify this somewhat. Requests for blocks in the heap come in various sizes. To make it possible to quickly satisfy a request (or determine that a request cannot be satisfied), it is useful to maintain a vector of free-lists — one for each of a set of size intervals. Logarithmically spaced size intervals are appropriate — e.g., one for each power of 2. To form the free-lists, the heap is swept linearly. Words marked as in use are ignored. A contiguous set of unmarked words is treated as a block. The number of words in the block is recorded in the head of each block, the number of leading zeros in the binary representation of the count is determined, and the block is entered into the appropriate free-list. Each block contains a pointer to the next block on that free-list.

blocks, each comprising contiguous words not marked as in use. The compactifying garbage collector moves all storage in use into one region of memory to produce a <u>single</u> <u>free</u> <u>block</u>.

Compactification begins by linearly sweeping the heap to form a free-list where each block contains a <u>count</u> <u>field</u> containing the number of words in the block and a pointer to the next block. Since the list is formed by a linear sweep, the list is ordered by address; c.f., Fig. 1.

The next phase involves <u>planning</u> where the blocks in use will be moved to. The desired rearrangement of storage blocks depends on system considerations. Two cases are perhaps most common. (1) Move all blocks in use to low core. Part of the resulting free block can then be returned to a higher level storage allocator, e.g., monitor. (2) Move blocks in use <u>away</u> from some region which must be expanded — e.g., the stack.

In either case, we assume that the blocks in use are to be packed into a contiguous region starting at location B′. Let B be the starting address of the heap before compactification. Let S_i be the number of words in the i^{th} free block. The compactification algorithm is somewhat simplified if every block in use, including the first, is preceded by a free block. This can be insured if a small <u>starting</u> <u>block</u> at location B is specially reserved. This will not be pointed to from any base point, so it will not be marked during tracing. It will be treated as free and, hence, merged with any other free words contiguous to it. Let S_0 be the size of the starting block and let S_1 be the size of the complete first free block (including the starting block). To initialize the bookkeeping so that the starting block remains reserved, the count field now containing S_1 is decremented by S_0.

With this convention, the address adjustment is simple. If the j^{th} block in use is now at U_j, its planned address, i.e., new address, will be (c.f., Fig. 2)

$$U'_j = U_j + (B'-B) - \sum_{i=1}^{j} S_i$$

The relevant variable is the relocation term R_j for the i^{th} block, i.e., the amount that the i^{th} block must be moved.

$$R_j = U'_j - U_j = (B'-B) - \sum_{i=1}^{j} S_i$$

It is clearly useful to set up a data structure which associates the term R_j with the j^{th} block in use. The only space guaranteed available is in the preceding free block. Hence, we use this space. The chain of free blocks is followed. In each free block, the count field is replaced by R_j. That is, S_1 is replaced by $(B'-B)-S_1$ and for all $j>1$, S_j is replaced by the new count field of the preceding block minus S_j. For example, assuming that $B'=B=1000$, we get the situation shown in Fig. 3.

The next phase of the compactifying garbage collector is to adjust each pointer to point to the planned address of the data object it references. That is, each pointer into the j^{th} block in use must have its word address increased by R_j. Note that the relocation term for a pointer depends only on which block it points to. Even if there are many distinct, separately allocated objects in a block, the relocation term for all pointers into the block is the same.

To carry out the pointer adjustment, the compactifier proceeds as follows. It clears the bit map, then performs a second trace. The second trace phase differs from the first in that every time a pointer is traced, it is adjusted to point to the planned location of the object it points to. Since pointers are being adjusted, it is necessary to insure that no pointer is traced twice in this process; the second trace of a pointer would result in a second, and hence erroneous, adjustment.

Therefore, the algorithm for the second trace phase differs somewhat from that for the first and requires several additional functions. ALL_MARKED(A,M) does not mark the object referenced by A and M but merely returns true if the object was previously marked. ALL_CLEAR(A,M) performs no marking and returns true if no words of the object are marked. MARK(A,M) marks an object but returns no value. HEADER_MARK(A,M) marks only the header words (if any) of a compound object; it leaves unchanged all words containing only the components. RELOC(A) returns the relocation term for the block which includes the address A; the implementation of this routine is discussed below.

To trace from a base point P, A and M are initialized as before, A is incremented by RELOC(A), and the following code is executed:

```
LOOP1:  if ALL_MARKED(A,M) then goto UP;
LOOP2:  if SIMPLE(M) then begin MARK(A,M); goto UP end;
        if POINTER(M) then begin MARK(A,M);
                                 A2 ← ADR(VAL(OBJ(A,M)));
                                 M2 ← MD(VAL(OBJ(A,M)));
                                 OBJ(A,M) ← OBJ(A,M)+RELOC(A2);
                                 A ← A2;
                                 M ← M2;
                                 goto LOOP1
                           end;
        if ALL_CLEAR(A,M) then begin MARK(A,M); FLAG ← 2 end
                           else begin HEADER_MARK(A,M); FLAG ← 1 end;
        I ← LENGTH(A,M);
        if I=0 then goto UP;
LOOP3:  if I>1 then PUSH(A,M,I-1,FLAG);
        A2 ← ADR(COMPONENT(OBJ(A,M), I));
        M2 ← MD(COMPONENT(OBJ(A,M),I));
        A ← A2;
        M ← M2;
        if FLAG = 1 then goto LOOP1 else goto LOOP2;
UP:     if EMPTY_STACK( ) then goto TRACE2_DONE;
        POP(A,M,I,FLAG);
        goto LOOP3
```

The second trace algorithm is similar to the first in the case of SIMPLE objects. POINTERs are relocated but otherwise treated as before. Compound objects such that no components have yet been considered are treated as before — i.e., the entire object is marked, thereby locking out entry when tracing other pointers. Compound objects, where one or more but not all components have been previously marked, are handled specially. The entire object is not marked; instead, a flag is set so that testing and marking are performed when the individual components are considered. The flag is stacked as part of the return point.

Notice what occurs now when two pointers, P1 and P2, respectively reference an object and one of its components. If P1 is traced first, the object is marked, correctly locking out the trace via P2. If P2 is traced first, the component is marked; hence, when tracing P1 the object is not found to be ALL_CLEAR. Only the header (if any) is marked and the flag is set. As each component of the object is considered, the algorithm tests whether that component is marked. The component referenced by P2 is found to be marked in this step, so it is not erroneously traced and relocated a second time.

This leaves the question of how to implement RELOC. Given a pointer to an address A, it is necessary to find the first free block immediately preceding A. Since the free blocks are linked, this could be carried out by searching the chain, a process which on the average would require half as many steps as free blocks. The search can be speeded up considerably with a directory of the free-list. Let the heap size be H and let the entries in the directory be designated 0, 1, 2, ... , k. Then the i^{th} element of the directory contains the address of the first free block in the address region $B+\lfloor H*i/k \rfloor$ to $B+\lfloor H*(i+1)/k \rfloor$ if such a block exists; otherwise, it contains the address of the last free block preceding $B+\lfloor H*i/k \rfloor$. Hence, given an address A, the $\lfloor (A-B)*k/H \rfloor$ entry of the directory gives the free list entry at which to start the search. If the free blocks are spread uniformly, this speeds up the search by a factor of k; a nonuniform spread results in a speed-up by some factor less than k.

The directory requires a contiguous block of storage for $k + 1$ entries. The larger the block, the better the speed-up in the modified free-list search. Any large block of storage available in the system may be pressed into service, e.g., unused stack space, I/0 buffers, etc. One candidate which will always be available is the heap itself. That is, the block for the directory can be constructed in the largest block in the free-list. It is necessary to keep only the count field and the field used for the free-list pointer — the rest can be used for the directory. This is attractive since it requires no extra storage, works under any conditions, but works faster when a large block is available. However, it should be recalled that compactification is generally triggered by failure of the normal garbage collection to find a sufficiently large block. Unless one attempts anticipatory compactification, compactification is the time when the desired large block will be unavailable.

An alternative to the directory is a binary tree constructed from the free blocks. Given an address of an object in a used block and a node in the tree, address arithmetic can be used to determine whether (1) the node covers the address (i.e., this is the appropriate free block for that address), (2) the up link of the node is to be followed to a node higher in the address space, or (3) the down link is to be followed to an entry lower in the address space. If the binary tree is complete and balanced, then with n free blocks, it will take at most $\lceil \log_2 n \rceil$ steps to find the appropriate node for any address. Constructing such a binary tree is straightforward. However, each node in the tree requires storage for two pointers and a covering address (in addition to a count field). The smallest free blocks may not be able to accommodate all these. Hence, a mixed strategy is called for. The binary tree can be built for the first $m < n$ choice levels only, its nodes constructed from the larger free blocks. Following, the binary tree yields either a node or a start point in the free-list from which point the list can be scanned by following the pointer chain.

The final phase is the actual <u>moving</u> of blocks. As all pointers have been corrected, it remains only to copy the blocks to their new locations. Since the copying is done by <u>blocks</u> of adjacent words in use, the number of distinct copy set-ups will be substantially smaller than the number of <u>objects</u> being handled. If $B' \leq B$, all the relocation terms will be negative and moving blocks is straightforward. The heap is swept linearly. The j^{th} block, at location U_j, is copied to $U_j + R_j$. Most machines have a "block transfer" or "move long" instruction for the actual copy operation. However, if $B' > B$ then one or more of the initial relocation terms, say $R_1 \ldots R_p$, may be positive. This will typically occur when the heap elements are being moved down in core to allow the stack to grow. The corresponding blocks must be moved in reverse order, p to 1, so as to prevent overwriting needed information. Further, some of the first blocks moved in this process must themselves be copied in reverse order (last word copied first), again so as to prevent overwriting.

When the block-moving phase is finished, the heap has been reorganized so that words in use form a contiguous block, pointers have been adjusted to reflect this reorganization, and non-pointer information remains unchanged. Compactifying garbage collection is therefore complete.

<u>Summary</u>

The technique presented in this paper emphasizes generality of application and parsimony of storage. It collects and compactifies structures and arrays of both pointer and non-pointer fields, requiring no reserved garbage collection fields. It requires only a single bit map and a stack of resumption points. It works correctly for arbitrary re-entrant structure, even where pointers point into components within an object. Where only individual components of an object are in use, it preserves these components and reclaims the words containing only the other components. Compactification is an option which requires no additional storage. When employed, it compacts all storage actually in use into a contiguous region, producing a single block of all free storage in the heap.

References

CHENEY, C. J. (1970). A Nonrecursive List Compacting Algorithm, Communications of the ACM, Vol. 13, pp. 677-678.

FENICHEL, R. R. and YOCHELSON, J. C. (1969). A LISP Garbage-Collector for Virtual-Memory Computer Systems, Communications of the ACM, Vol. 12, pp. 611-612.

HANSEN, W. J. (1969). Compact List Representation: Definition, Garbage Collection, and System Implementation, Communications of the ACM, Vol. 12, pp. 499-507.

MC CARTHY, J. et al. (1962). Lisp 1.5 Programmer's Manual. Cambridge, Massachusetts: The M. I. T. Press.

MINSKY, M. L. (1963). A LISP Garbage Collector Algorithm Using Serial Secondary Storage, M. I. T. Artificial Intelligence Project, Memo 58, Cambridge, Massachusetts.

SCHORR, H. and WAITE, W. M. (1967). An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, Communications of the ACM, Vol. 10, pp. 501-506.

WEGBREIT, B. (1971). The ECL Programming System, Proceedings Fall Joint Computer Conference 1971, Vol. 39.

WIJNGAARDEN, A. V. et al. (1969). Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 101.

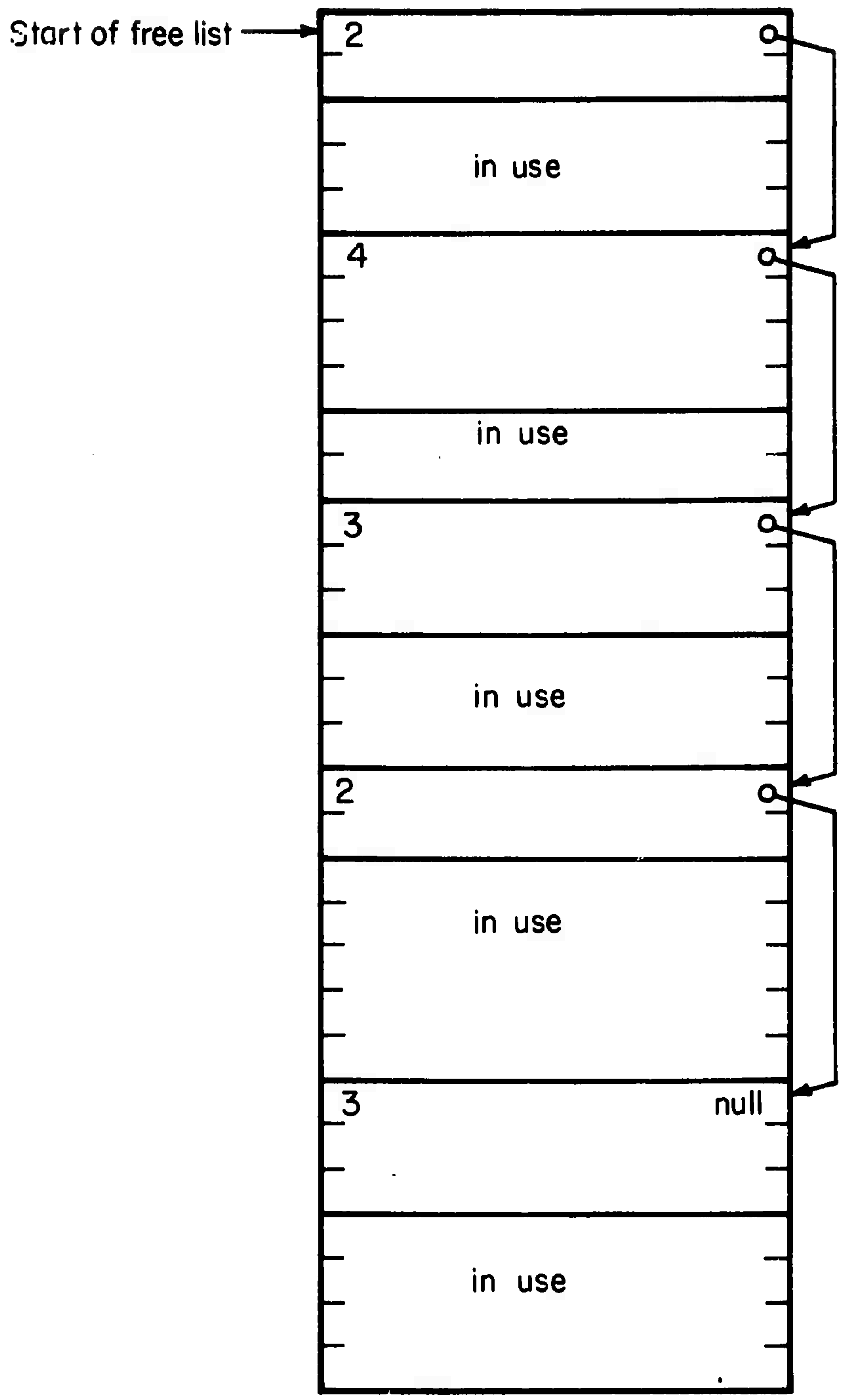

Fig. 1 Free list

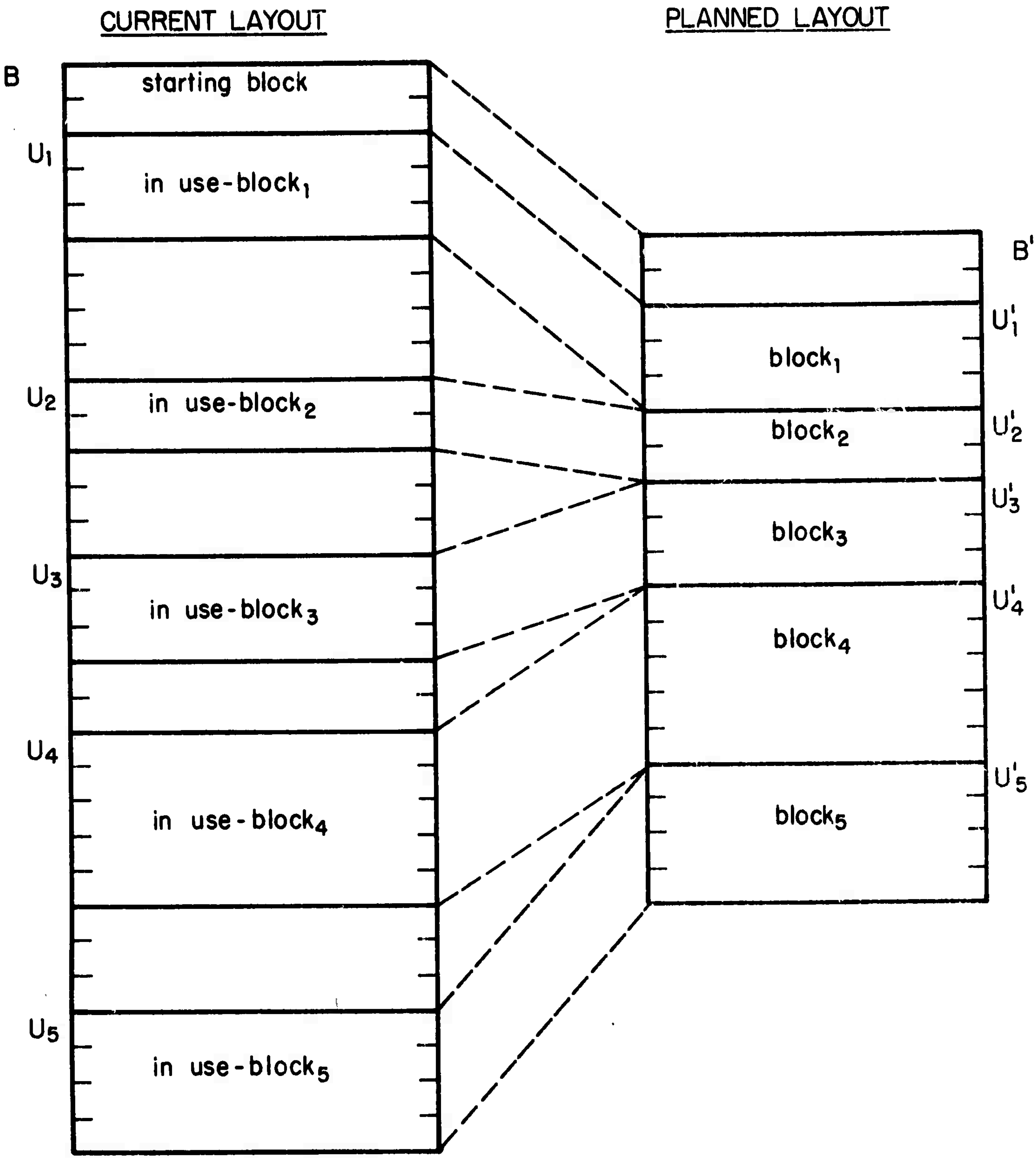

Fig. 2 Planned compactification

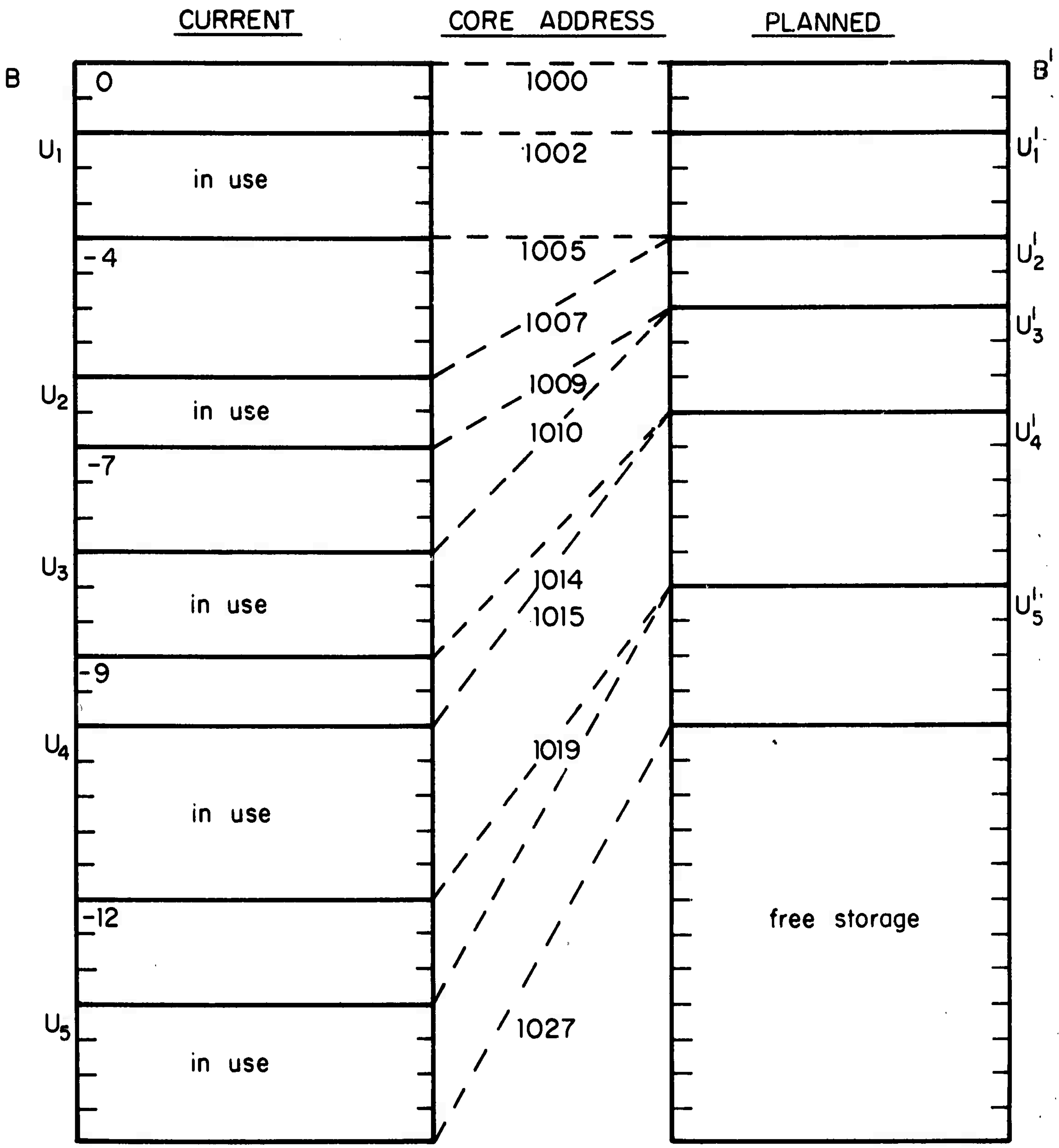

Fig 3 Example with $B = B'$